IE2061 - Operating Systems and System Administration

Lecture 05 : Introduction to Threads

SLIIT FACULTY OF COMPUTING

Motivation

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

Threads

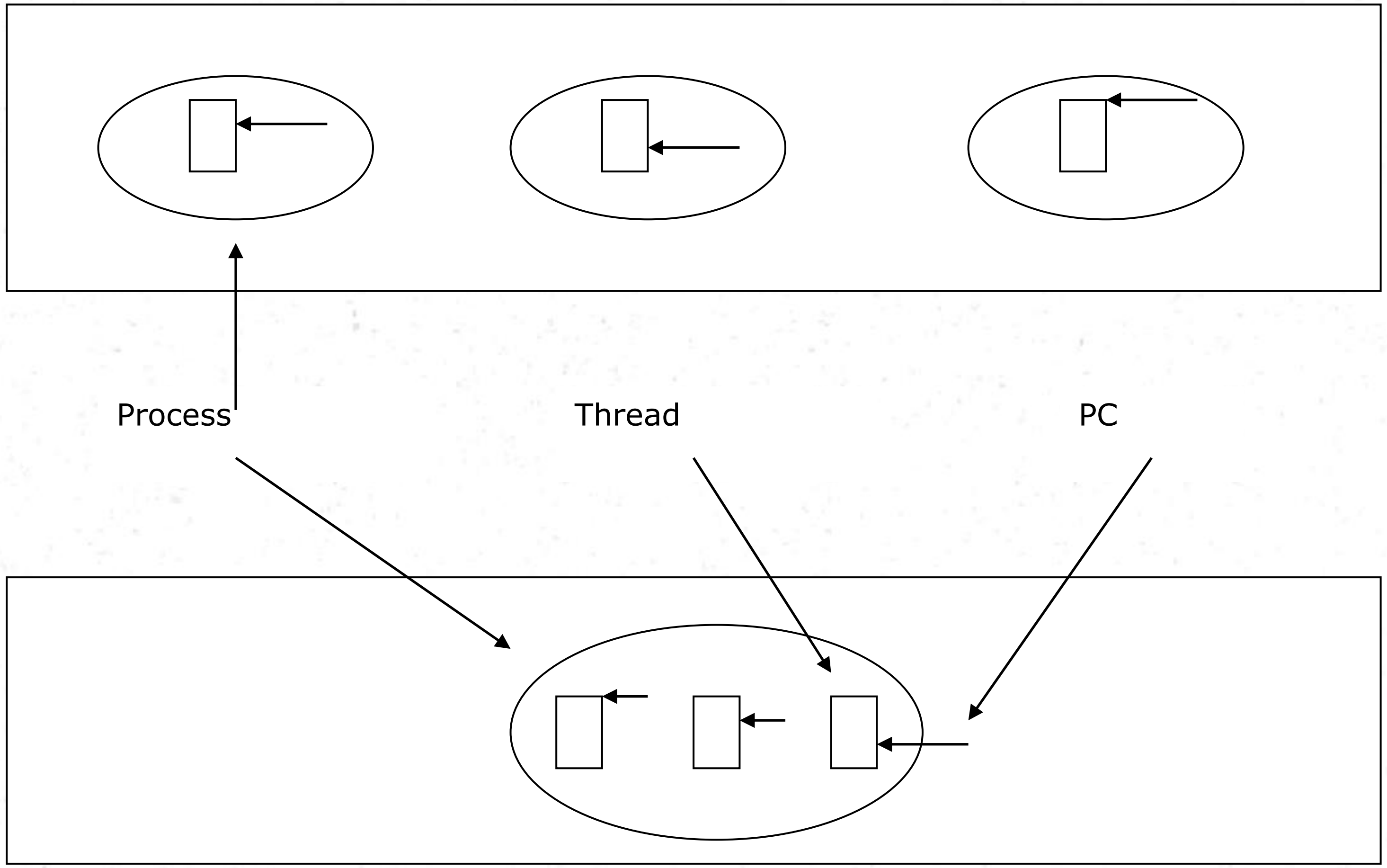
- A thread of control is an independent sequence of execution of program code in a process.
- A thread (or lightweight process) is a basic unit of CPU utilization.
  - A traditional process (or heavyweight process) is equal to a task with one thread.
- A traditional process has a single thread that has sole possession of the process's memory and other resources → context switch becomes performance bottleneck
  - Threads are used to avoid the bottleneck.
  - Threads share all the process's memory, and other resources.
- Threads within a process:
  - are generally invisible from outside the process.
  - are scheduled and executed independently in the same way as different single-threaded processes.
- On a multiprocessor, different threads may execute on different processors
  - On a uni-processor, threads may interleave their execution arbitrarily.

Threads (cont.)

- Threads operate, in many respects, in the same manner as processes:
  - Threads can be in one of several states: ready, blocked, running, or terminated, etc.
  - Threads share CPU; only one thread at a time is running.
  - A thread within a process executes sequentially, and each thread has its own PC and Stack.
  - Thread can create child threads, can block waiting for system calls to complete
    - if one thread is blocked, another can run.
- One major different with process: threads are not independent of one another
  - all threads can access every address in the task → a thread can read or write any other thread's stack.
  - There is no protection between threads (within a process);
    - however this should not be necessary since processes may originate from different users and may hostile to one another while threads (within a process) should be designed (by same programmer) to assist one another.

Threads (cont.)

Three processes with one thread each

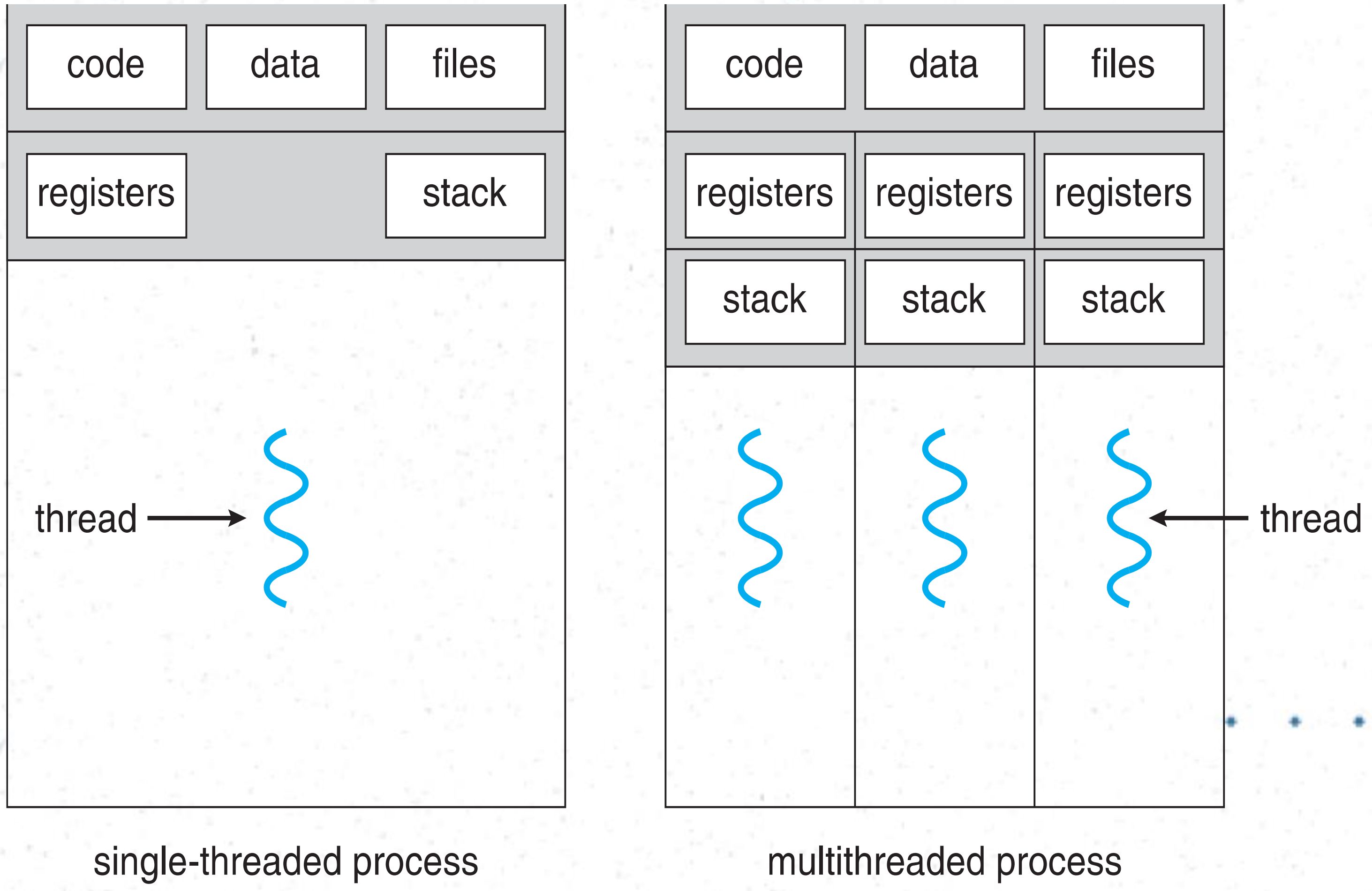


One process with three threads

Benefits

- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing
- **Economy –** cheaper than process creation, thread switching lower overhead than context switching
- **Scalability –** process can take advantage of multiprocessor architectures

Single and Multithreaded Processes



User Threads and Kernel Threads

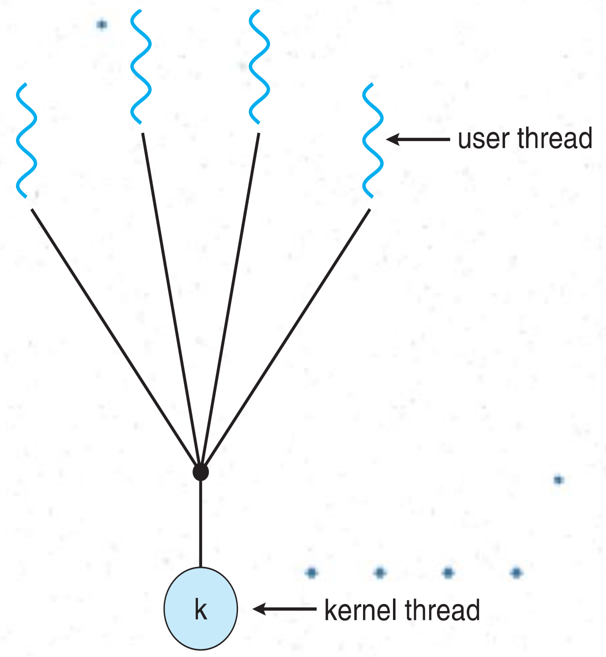
- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

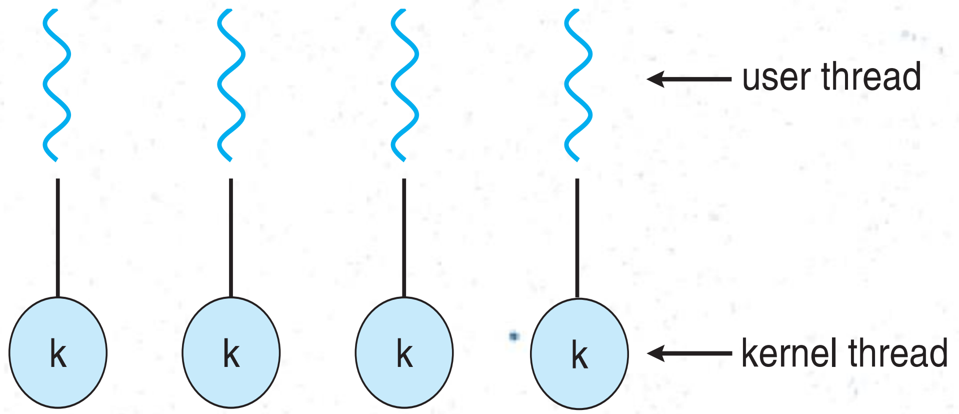
Many-to-One

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**



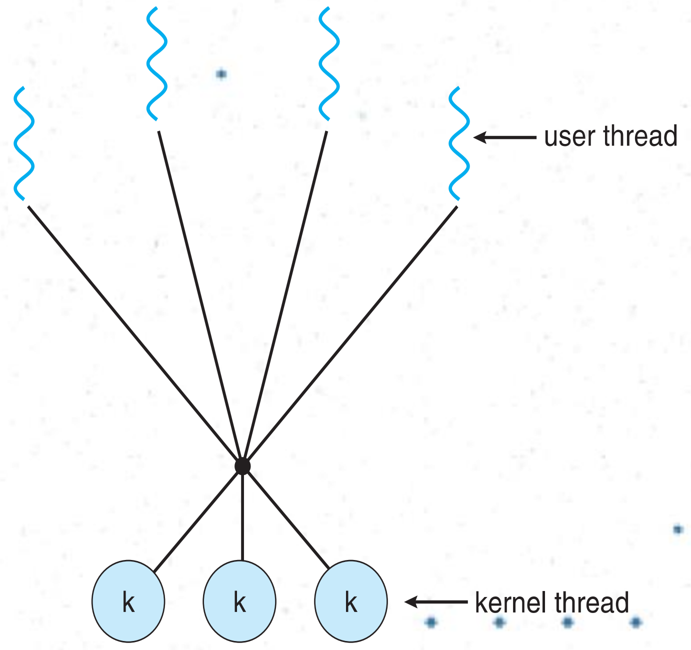
One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later



Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the *ThreadFiber* package



Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification***, not ***implementation***
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Thread Pools

- Create a number of threads in a pool where they await work
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool
  - Separating task to be performed from mechanics of creating task allows different strategies for running task
    - i.e.Tasks could be scheduled to run periodically

Signal Handling

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
- A **signal handler** is used to process signals
  - Signal is generated by particular event
  - Signal is delivered to a process
  - Signal is handled by one of two signal handlers:
    - default
    - user-defined
- Every signal has **default handler** that kernel runs when handling signal
  - **User-defined signal handler** can override default
  - For single-threaded, signal delivered to process

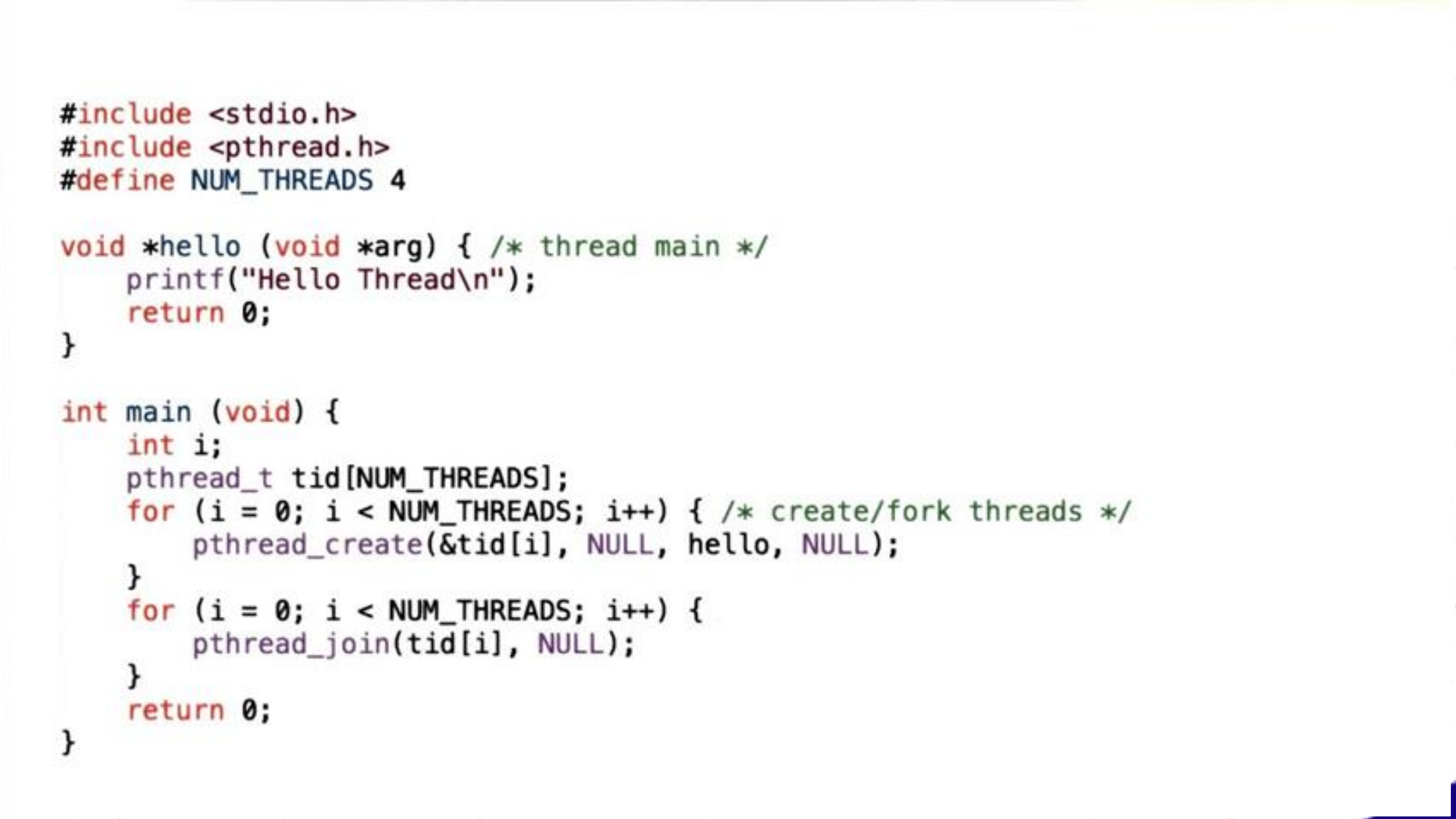
Signal Handling (Cont.)

- Where should a signal be delivered for multi-threaded?
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process

Thread Cancellation

- Terminating a thread before it has finished
- Thread to be canceled is **target thread**
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled

Thread Example



```
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS 4

void *hello (void *arg) { /* thread main */
    printf("Hello Thread\n");
    return 0;
}

int main (void) {
    int i;
    pthread_t tid[NUM_THREADS];
    for (i = 0; i < NUM_THREADS; i++) { /* create/fork threads */
        pthread_create(&tid[i], NULL, hello, NULL);
    }
    for (i = 0; i < NUM_THREADS; i++) {
        pthread_join(tid[i], NULL);
    }
    return 0;
}
```

```
#include<pthread.h> #include <stdio.h>
/* Producer/consumer program illustrating conditional variables */ /* Size of shared buffer */       #define BUF_SIZE 3
int buffer[BUF_SIZE];                                                    /* shared buffer */
int add=0;                                                               /* place to add next element */
int rem=0;                                                               /* place to remove next element */
int num=0;                                                               /* number elements in buffer */
pthread_mutex_t m=PTHREAD_MUTEX_INITIALIZER;             /* mutex lock for buffer */
pthread_cond_t c_cons=PTHREAD_COND_INITIALIZER; /* consumer waits on this cond var */
pthread_cond_t c_prod=PTHREAD_COND_INITIALIZER; /* producer waits on this cond var */
void *producer(void *param);
void *consumer(void *param);
main (int argc, char *argv[])
{               pthread_t tid1, tid2;                    /* thread identifiers */
                int i; /* create the threads; may be any number, in general */
                if (pthread_create(&tid1,NULL,producer,NULL) != 0) {
                                fprintf (stderr, "Unable to create producer thread\n"); exit (1); }
                if (pthread_create(&tid2,NULL,consumer,NULL) != 0) {
                                fprintf (stderr, "Unable to create consumer thread\n"); exit (1);
                }
                /* wait for created thread to exit */
                pthread_join(tid1,NULL);
                pthread_join(tid2,NULL);
                printf ("Parent quiting\n"); }
}
```

```
/* Produce value(s) */
void *producer(void *param)
{
          int i;
          for (i=1; i<=20; i++) {
                    /* Insert into buffer */
                    pthread_mutex_lock (&m);
                    if (num > BUF_SIZE) exit(1);          /* overflow */
                    while (num == BUF_SIZE)                              /* block if buffer is full */
                              pthread_cond_wait (&c_prod, &m);
                    /* if executing here, buffer not full so add element */
                    buffer[add] = i;
                    add = (add+1) % BUF_SIZE;
                    num++;
                    pthread_mutex_unlock (&m);
                    pthread_cond_signal (&c_cons);
                    printf ("producer: inserted %d\n", i);  fflush (stdout);
          }
          printf ("producer quiting\n");  fflush (stdout);
•   }
```

```
/* Consume value(s); Note the consumer never terminates */
void *consumer(void *param)
{
        int i;
        while (1) {
                pthread_mutex_lock (&m);
                if (num < 0) exit(1);   /* underflow */
                while (num == 0)                /* block if buffer empty */
                        pthread_cond_wait (&c_cons, &m);
                /* if executing here, buffer not empty so remove element */
                i = buffer[rem];
                rem = (rem+1) % BUF_SIZE;
                num--;
                pthread_mutex_unlock (&m);
                pthread_cond_signal (&c_prod);
                printf ("Consume value %d\n", i);  fflush(stdout);
        }
}
```

End of Lecture 5

IT2060 - Operating Systems and System Administration

Lecture 04 : CPU Scheduling

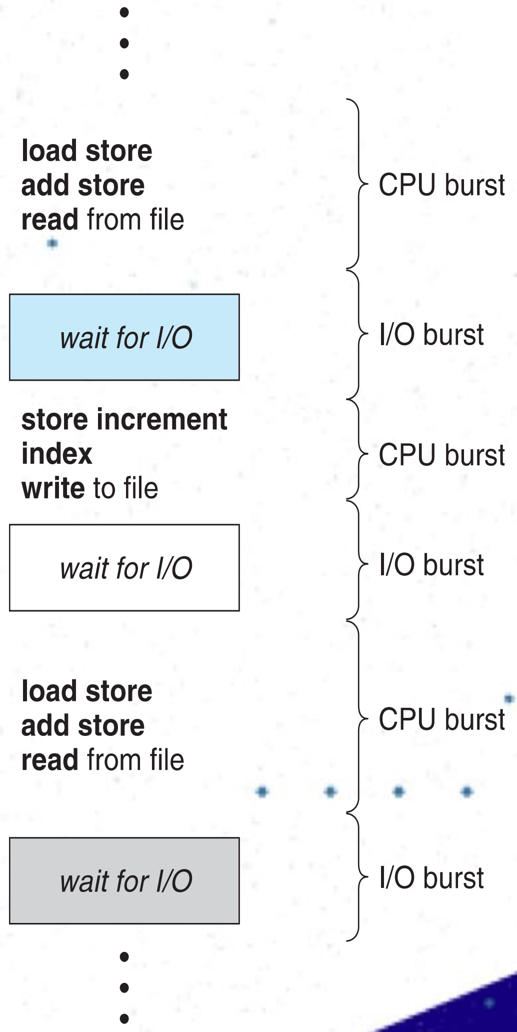
SLIIT
FACULTY OF COMPUTING

CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Algorithm Evaluation

Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern



CPU Scheduler

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running

Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

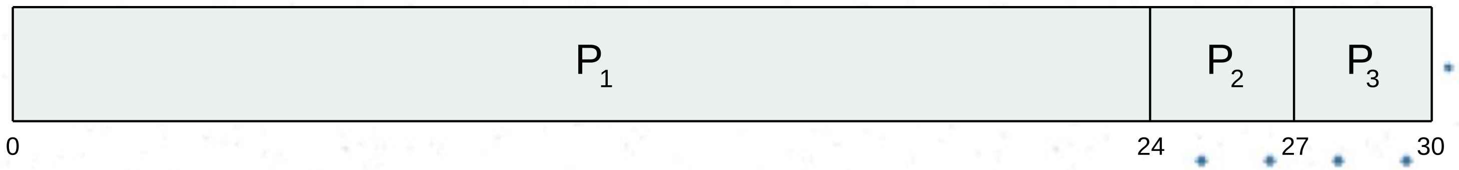
Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

First- Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:



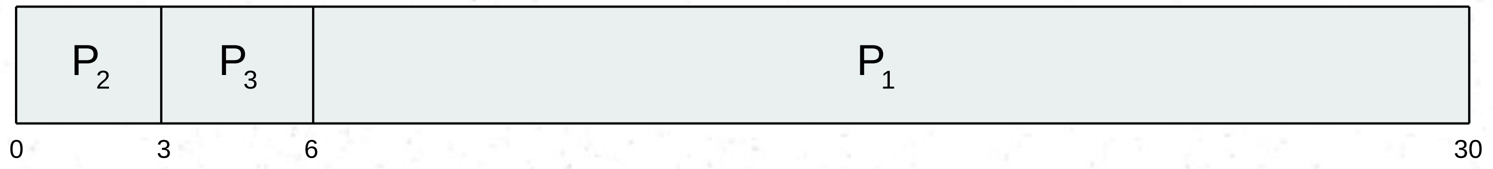
- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: (0 + 24 + 27)/3 = 17

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

P_2 , P_3 , P_1

- The Gantt chart for the schedule is:



- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case
- **Convoy effect** - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

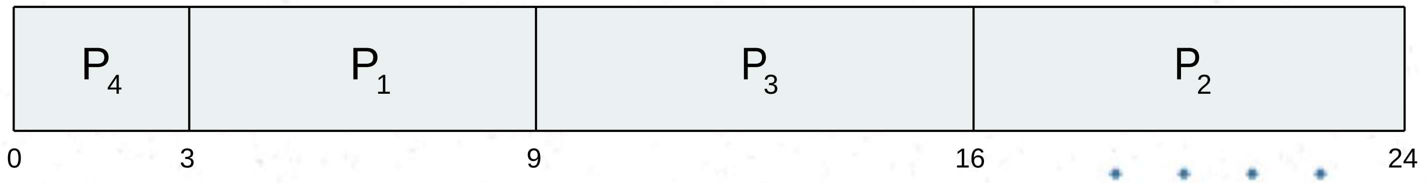
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request
  - Could ask the user

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



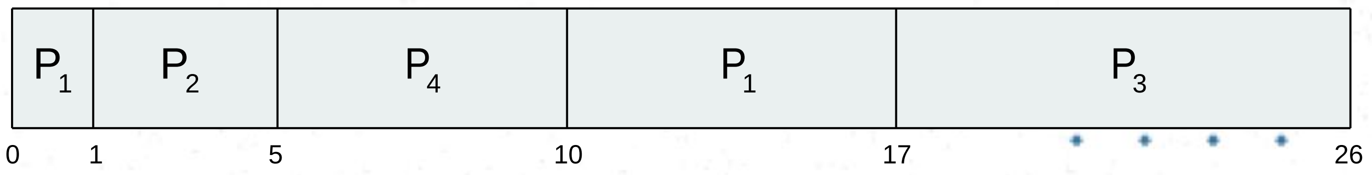
- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

Example of Shortest-remaining-time-first

- Now we add the concepts of varying arrival times and preemption to the analysis

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- *Preemptive* SJF Gantt Chart



- Average waiting time = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 msec

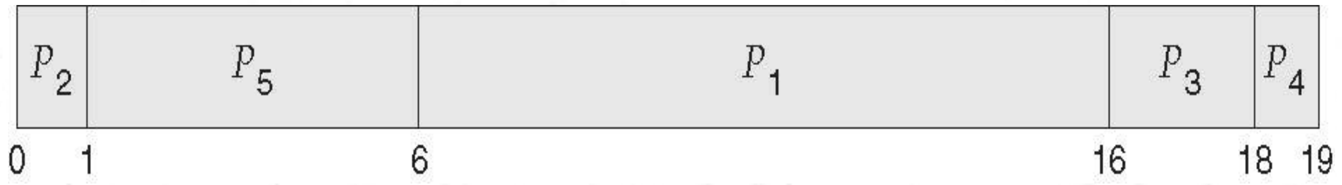
Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process

Example of Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart



- Average waiting time = 8.2 msec

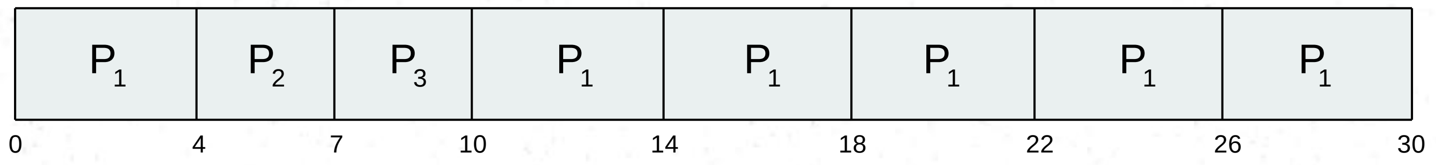
Round Robin (RR)

- Each process gets a small unit of CPU time (**time quantum** *q*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than (*n*-1)*q* time units.
- Timer interrupts every quantum to schedule next process
- Performance
  - *q* large $\Rightarrow$ FIFO
  - *q* small $\Rightarrow$ *q* must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 4

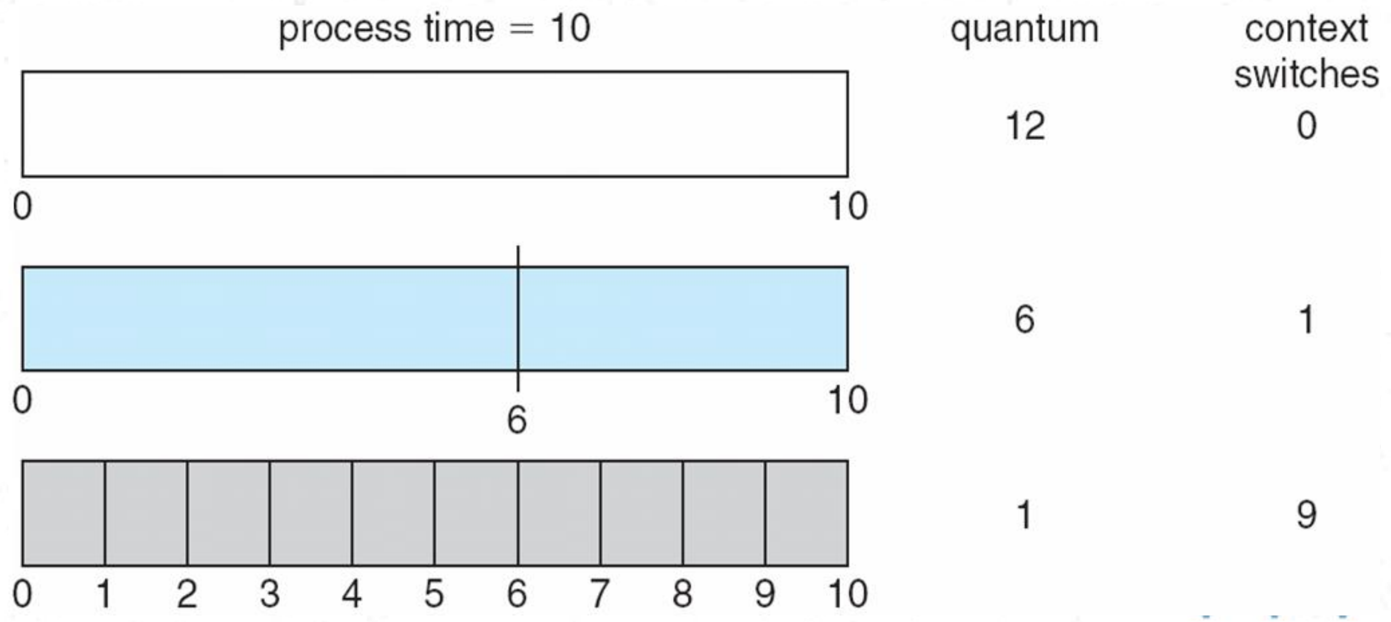
Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 usec

Time Quantum and Context Switch Time



Multilevel Queue

- Ready queue is partitioned into separate queues, eg:
  - **foreground** (interactive)
  - **background** (batch)
- Process permanently in a given queue
- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR 20% to background in FCFS

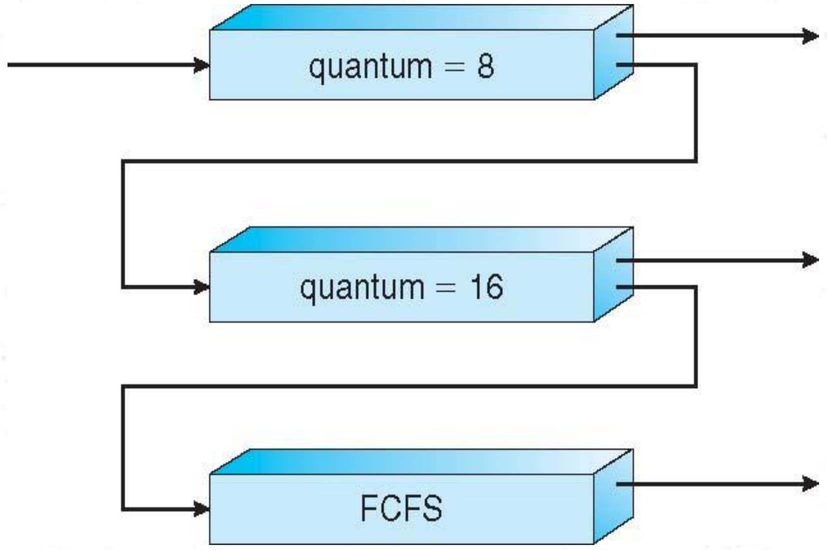
Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS

- Scheduling
  - A new job enters queue Q_0 which is served FCFS
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue Q_1
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2



Algorithm Evaluation

- How to select CPU-scheduling algorithm for an OS?
- Determine criteria, then evaluate algorithms
- **Deterministic modeling**
  - Type of **analytic evaluation**
  - Takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Consider 5 processes arriving at time 0:

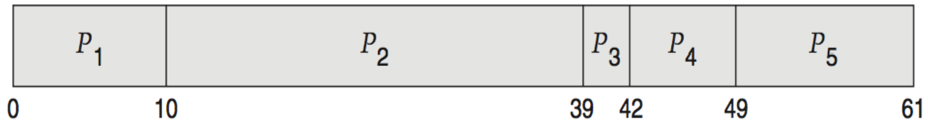
Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

Deterministic Evaluation

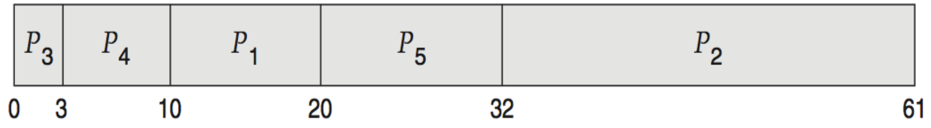
For each algorithm, calculate minimum average waiting time

Simple and fast, but requires exact numbers for input, applies only to those inputs

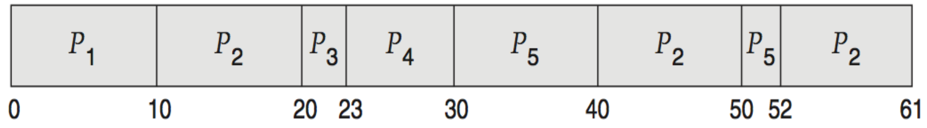
- FCS is 28ms:



- Non-preemptive SFJ is 13ms:



- RR is 23ms:



Queueing Models

- Describes the arrival of processes, and CPU and I/O bursts probabilistically
  - Commonly exponential, and described by mean
  - Computes average throughput, utilization, waiting time, etc
- Computer system described as network of servers, each with queue of waiting processes
  - Knowing arrival rates and service rates
  - Computes utilization, average queue length, average wait time, etc

Little's Formula

- n = average queue length
- W = average waiting time in queue
- λ = average arrival rate into queue
- Little's law – in steady state, processes leaving queue must equal processes arriving, thus:

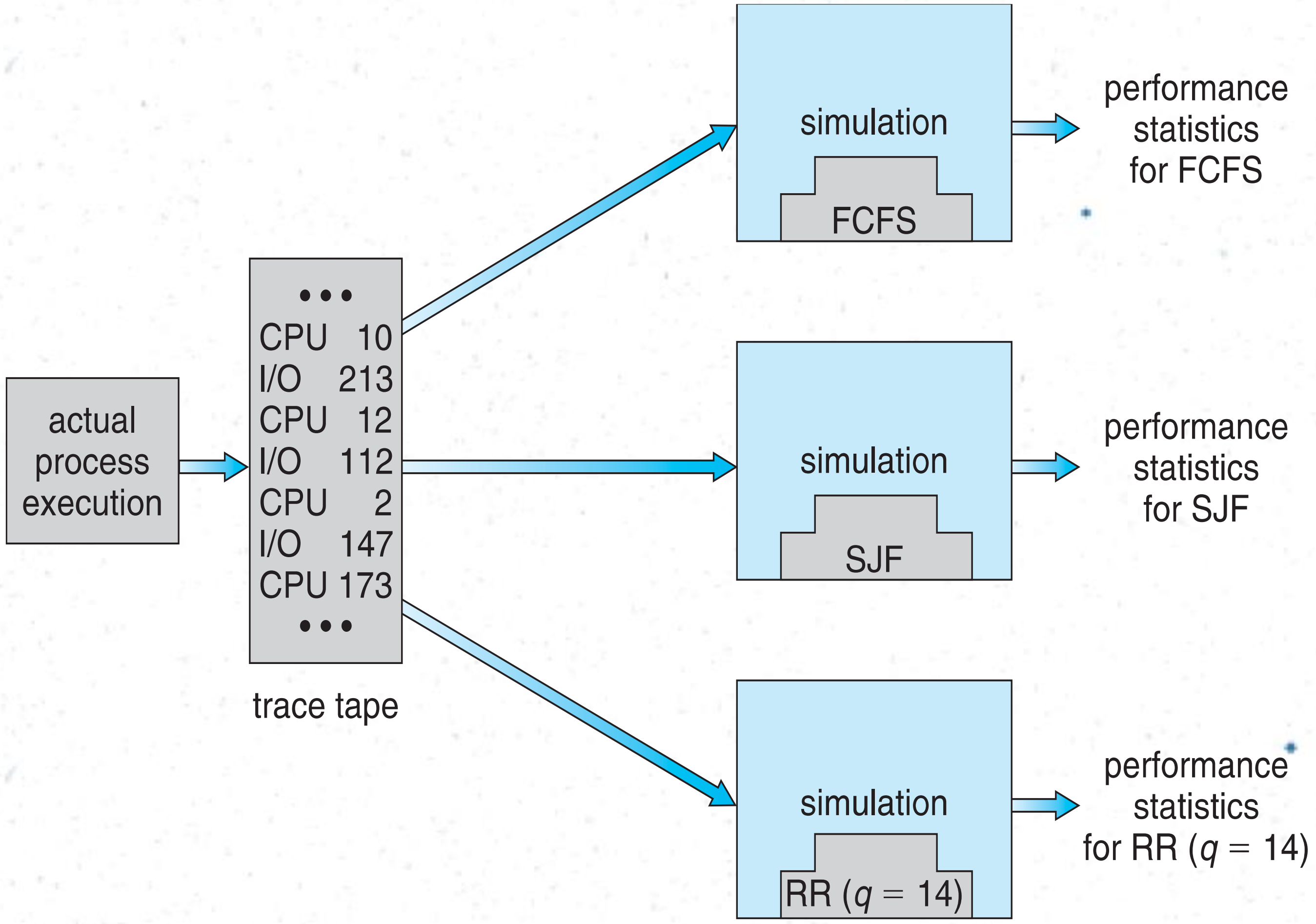
  $$n = \lambda \times W$$

  - Valid for any scheduling algorithm and arrival distribution
- For example, if on average 7 processes arrive per second, and normally 14 processes in queue, then average wait time per process = 2 seconds

Simulations

- Queueing models limited
- **Simulations** more accurate
  - Programmed model of computer system
  - Clock is a variable
  - Gather statistics indicating algorithm performance
  - Data to drive simulation gathered via
    - Random number generator according to probabilities
    - Distributions defined mathematically or empirically
    - Trace tapes record sequences of real events in real systems

Evaluation of CPU Schedulers by Simulation



Implementation

- Even simulations have limited accuracy
- Just implement new scheduler and test in real systems
  - High cost, high risk
  - Environments vary
- Most flexible schedulers can be modified per-site or per-system
- Or APIs to modify priorities
- But again environments vary

End of Lecture 4

IT2060 - Operating Systems and System Administration

Lecture 06 : Process Synchronization

SLIIT FACULTY OF COMPUTING

Chapter 5: Process Synchronization

- Background
- The Critical-Section Problem
- Semaphores
- Classic Problems of Synchronization
- Monitors

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem:
  Suppose that we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffers. We can do so by having an integer **`counter`** that keeps track of the number of full buffers. Initially, **`counter`** is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer

```
while (true) {
        /* produce an item in next produced */

        while (counter == BUFFER_SIZE) ;
                /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

Consumer

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
         counter--;
        /* consume the item in next consumed */
}
```

Race Condition

- `counter++` could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- `counter--` could be implemented as

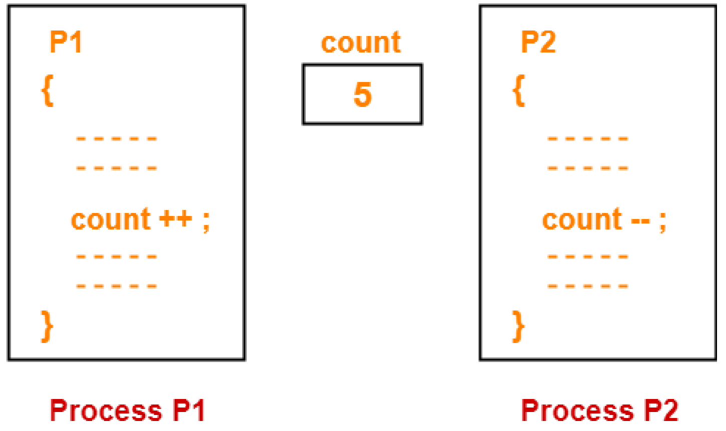
```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with "count = 5" initially:

```
S0: producer execute register1 = counter           {register1 = 5}
S1: producer execute register1 = register1 + 1     {register1 = 6}
S2: consumer execute register2 = counter           {register2 = 5}
S3: consumer execute register2 = register2 – 1     {register2 = 4}
S4: producer execute counter = register1           {counter = 6 }
S5: consumer execute counter = register2           {counter = 4}
```

Race condition

- *Race condition* is a situation where several processes access and manipulate the same data concurrently.
  - The outcome of the execution depends on particular order in which the access takes place.
- In order to prevent race condition on *counter*, we need to ensure that only one process at a time can be manipulating *counter*
  - We need some form of *process synchronization*.

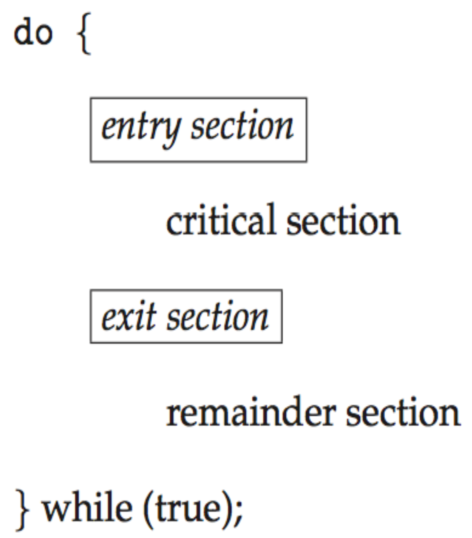


Critical Section Problem

- Consider system of ***n*** processes {p_0, p_1, … p_{n-1}}
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

- General structu



```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process ***P**$_i$* is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

Solution (cont.)

Simplest solution:

Each process *disables* all interrupts just *after entering* its critical section and *re-enables* them just *before leaving* it.

- Not wise, because enabling/disabling interrupt is a privileged instruction.

Solution in kernel mode:

- **Preemptive kernel:** a process can be preempted while running in the kernel mode
  - Otherwise, the kernel is **nonpreemptive kernel**: allows the process to run until it exits kernel mode, blocks, or voluntarily yields CPU.
- Nonpreemptive kernel is free from race conditions on kernel data structure
  - However preemptive kernel is more responsive and suitable for real time system.

Solution for two processes, P_0 and P_1

Software-based

ALGORITHM 1

var *turn*: (0 .. 1); // Initially turn = 0; *turn* = *i* means P_i can enter its CS

```
Process P_i                                Process P_j
repeat                                     repeat
    while turn ≠ i do no-op;               while turn ≠ j do no-op;
            Critical Section                       Critical Section
    turn = j;                                  turn = i;
            Remainder Section              Remainder Section
until false;                               until false;
```

- Satisfies mutual exclusion, but not progress requirement.
  - If *turn* = 0, P_1 cannot enter its CS even though P_0 is in its RS.
  - Taking turn is not good when one process is slower than other.
- *Busy waiting*: continuously testing a variable waiting for some value to appear
  - not good since it wastes CPU time

Solution for two processes (cont.)

Software-based

ALGORITHM 2

// Initially *flag*[0] = *flag*[1] = *false*; *flag[i]* = *true* means P_i wants to enter its CS

var *flag*: **array** [0 .. 1] **of** *boolean;*

```
Process P_i                                    Process P_j
repeat                                    repeat
   flag[i] = true;                             flag[j] = true;
   while flag[j] do no-op;        while flag[i] do no-op;
            Critical Section;                          Critical Section;
   flag[i] = false;               flag[j] = false;
            Remainder Section;                         Remainder Section;
until false;                              until false;
```

- Satisfy mutual exclusion, but violates the progress requirement:
  T_0: P_0 sets flag[0] = true.
  T_1: P_1 sets flag[1] = true.
  → P_0 and P_1 are looping in their respective while.

Solution for two processes (cont.)

Software-based

// **Peterson's solution**: Combine shared variables of Algorithms 1 and 2

Process P_i

```
repeat
    flag[i] = true;
    turn = j;
    while (flag[j] and turn = j) do no-op;
            critical section
    flag[i] = false;
            remainder section
until false;
```

Process P_j

```
repeat
    flag[j] = true;
    turn = i;
    while (flag[i] and turn =i) do no-op;
            critical section
    flag[j] = false;
            remainder section
until false;
```

- Solves the critical-section problem for two processes.
    - It meets all the three requirements
- Proof: need to show that:
    - Mutual exclusion is preserved.
    - The progress requirement is satisfied.
    - The bounded waiting time requirement is met.
- For detailed proof, read the textbook.

Bakery Algorithm

- The solution to the critical section problem for *n* processes by Leslie Lamport
- Before entering its critical section, each process receives a number.
    - The holder of the smallest number enters the critical section.
    - If processes P_i and P_j receive the same number, if $i < j$, then P_i is served first; else P_j is served first.
- Notation (**ticket#**, process **id#**)
    - $(a, b) < (c, d)$ if $a < c$ or if $a = c$ and $b < d$.
    - $max\ (a_0, \ldots, a_{n-1})$ is a number k such that $k \geq a_i$ for $i = 0, \ldots, n-1$.
- shared data
    - **var** *choosing*: **array**[0..*n*-1] of *boolean*;
    - *number*: **array** [0 .. *n*-1] of *integer*;
- data structures are initialised to *false* and 0, respectively

Bakery Algorithm (contd.)

Process P_i

```
repeat
    choosing[i] = true;
    number[i] = max (number[0], number[1], …, number[n-1]) +1;
    choosing[i] = false;
    for j = 0 to n-1 do
    begin
        while choosing[j] do no-op;
          while number[j] ≠ 0 and (number[j],  j) < (number[i], i) do no-op;
    end;
            critical section
    number[i] = 0;
            remainder section
until false
```

Synchronization Hardware

- There is no guarantee that the software-based solution will work correctly in all computer architectures.
- The simple solution to critical section problem: disable interrupt while a shared variable is being modified.
  - This solution is not feasible in multiprocessor. Why?
- Use special hardware instructions such as *Test-and-Set* and *Swap.*
  - *Test-and-set* or *Swap* is an atomic instruction: it can not be interrupted until it completes its execution

```
// Test and set the content of a word
   atomically

function Test-and-Set (var boolean: target)
begin
    Test-and-Set = target;
    target = true;
end;
```

```
// Swapping instruction is done
   atomically

procedure Swap (var boolean: a, b)
var boolean: temp;
begin
    temp = a;
    a = b;
    b = temp;
end;
```

How to use them?

Mutual Exclusion with *Test-and-Set*

var *boolean*: *lock*; *lock* is a shared variable, initially set to *false*.

```
Repeat // Process $P_i$
    while Test-and-Set (lock) do no-op;
            Critical Section
    lock = false;
            Remainder Section
until false;
```

```
Repeat // Process $P_j$
    while Test-and-Set (lock) do no-op;
            Critical Section
    lock = false;
                    Remainder Section
until false;
```

Mutual Exclusion with *Swap*

```
Repeat  // Process $P_i$
    key = true;
    repeat
            Swap (lock, key);
    until key = false;
            Critical section
    lock = false;
            Remainder section
until false;
```

```
Repeat  // Process $P_j$
        key = true;
        repeat
            Swap (lock, key);
        until key = false;
            Critical section
        lock = false;
            Remainder section
until false;
```

- Both do not satisfy the bounded waiting requirement.

Correct solution with Test-and-set

Shared data: **var** *waiting*: **array**[0..*n*-1] **of** *boolean*; *lock*: *boolean*; //All initialized to *false*

Process P_i

```
var  j: 0..n-1;  key: boolean;
repeat
    waiting[i] = true;
    key = true;
    while waiting [i] and key do          // enter CS if either waiting[i] or key is false
            key = Test-and-Set (lock);    // key is false if lock is false
    waiting[i] = false;
            Critical Section
    j = i+1 mod n;
    while (j ≠ i) and not waiting[j] do  // check if any P_j is waiting for CS
            j = j+1 mod n
     if j = i then
            lock = false;                // no other process is waiting for CS
     else
             waiting[j] = false;         // P_j is waiting, let it enter CS next

             Remainder Section
until false;
```

Proof: Read textbook.

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
    - Originally called **`P()`** and **`V()`**
- Definition of the **`wait() operation`**

```
wait(S) {
    while (S <= 0)
       ; // busy wait
    S--;
}
```

- Definition of the **`signal() operation`**

```
signal(S) {
    S++;
}
```

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can solve various synchronization problems
- Consider $\boldsymbol{P_1}$ and $\boldsymbol{P_2}$ that require $\boldsymbol{S_1}$ to happen before $\boldsymbol{S_2}$

  Create a semaphore "**synch**" initialized to 0

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

- Can implement a counting semaphore ***S*** as a binary semaphore

Semaphore Implementation

- Must guarantee that no two processes can execute the `wait()` and `signal()` on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the `wait` and `signal` code are placed in the critical section
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
    - value (of type integer)
    - pointer to next record in the list
- Two operations:
    - **block** – place the process invoking the operation on the appropriate waiting queue
    - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue
-
```
typedef struct{
int value;
struct process *list;
} semaphore;
```

Implementation with no Busy waiting (Cont.)

```
wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      add this process to S->list;
      block();
   }
}

signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
      remove a process P from S->list;
      wakeup(P);
   }
}
```

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Bounded-Buffer Problem

- *n* buffers, each can hold one item
- Semaphore `mutex` initialized to the value 1
- Semaphore `full` initialized to the value 0
- Semaphore `empty` initialized to the value n

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do {
       ...
      /* produce an item in next_produced */
       ...
    wait(empty);
    wait(mutex);
        ...
      /* add next produced to the buffer */
        ...
    signal(mutex);
    signal(full);
 } while (true);
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
Do {
    wait(full);
    wait(mutex);
        ...
    /* remove an item from buffer to next_consumed */
        ...
    signal(mutex);
    signal(empty);
        ...
    /* consume the item in next consumed */
        ...
 } while (true);
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do ***not*** perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered – all involve some form of priorities
- Shared Data
  - Data set
  - Semaphore `rw_mutex` initialized to 1
  - Semaphore `mutex` initialized to 1
  - Integer `read_count` initialized to 0

Readers-Writers Problem (Cont.)

- The structure of a writer process

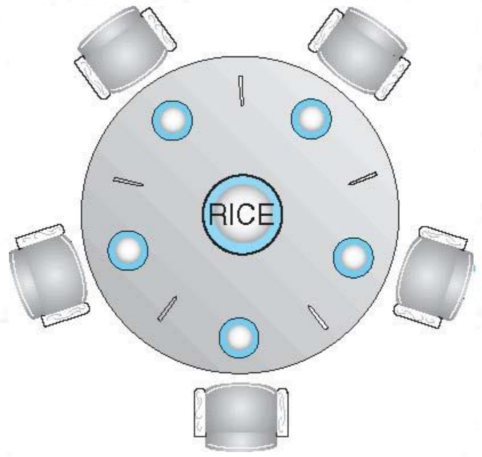
```
    do {
            wait(rw_mutex);
                ...
            /* writing is performed */
                ...
         signal(rw_mutex);
 } while (true);
```

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {
      wait(mutex);
      read_count++;
      if (read_count == 1)
       wait(rw_mutex);
    signal(mutex);
        ...
      /* reading is performed */
        ...
    wait(mutex);
      read_count--;
      if (read_count == 0)
    signal(rw_mutex);
    signal(mutex);
} while (true);
```

Dining-Philosophers Problem



- Philosophers spend their lives alternating thinking and eating
- Don’t interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
    - Need both to eat, then release both when done
- In the case of 5 philosophers
    - Shared data
        - Bowl of rice (data set)
        - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
do {
     wait (chopstick[i] );
      wait (chopStick[ (i + 1) % 5] );

                    //  eat

      signal (chopstick[i] );
      signal (chopstick[ (i + 1) % 5] );

                     //  think


 } while (TRUE);
```

- What is the problem with this algorithm?

Dining-Philosophers Problem Algorithm (Cont.)

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section.
  - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

Problems with Semaphores

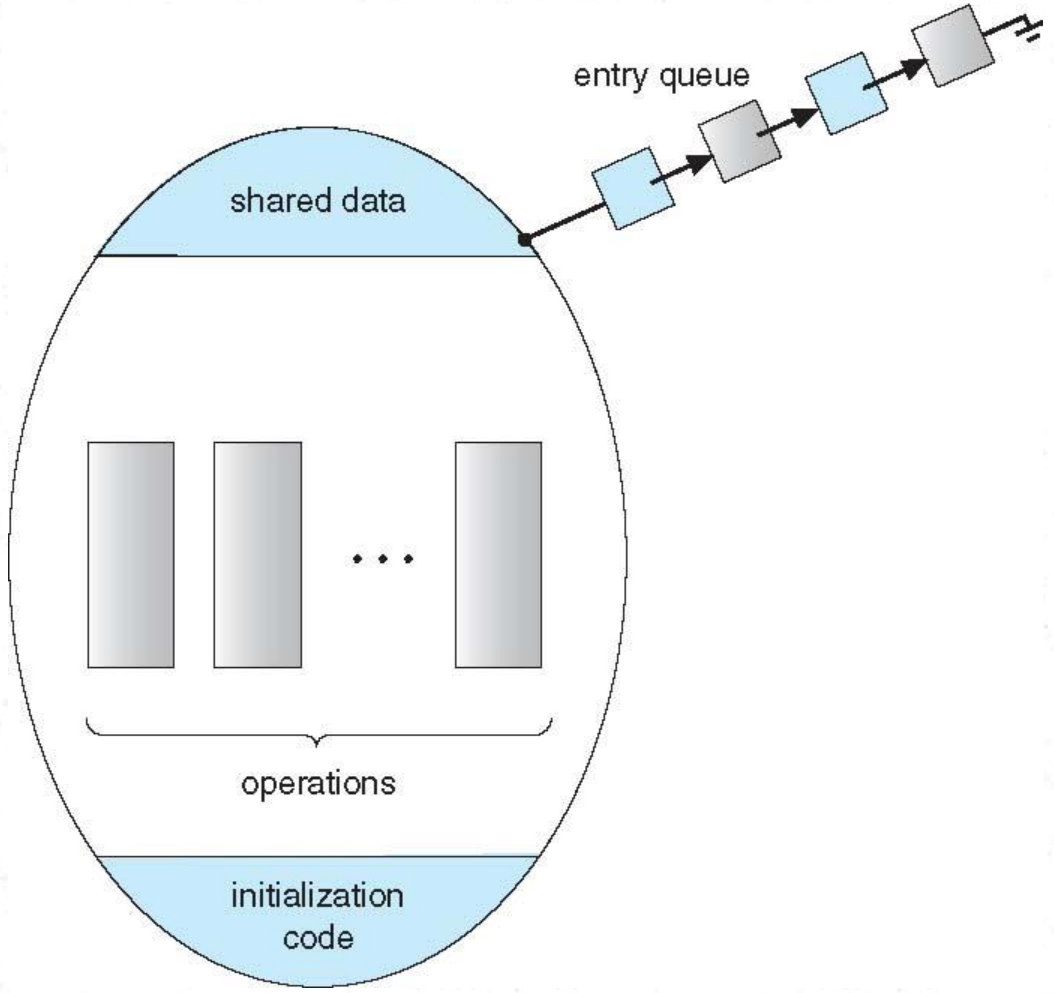
- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation are possible.

Monitors

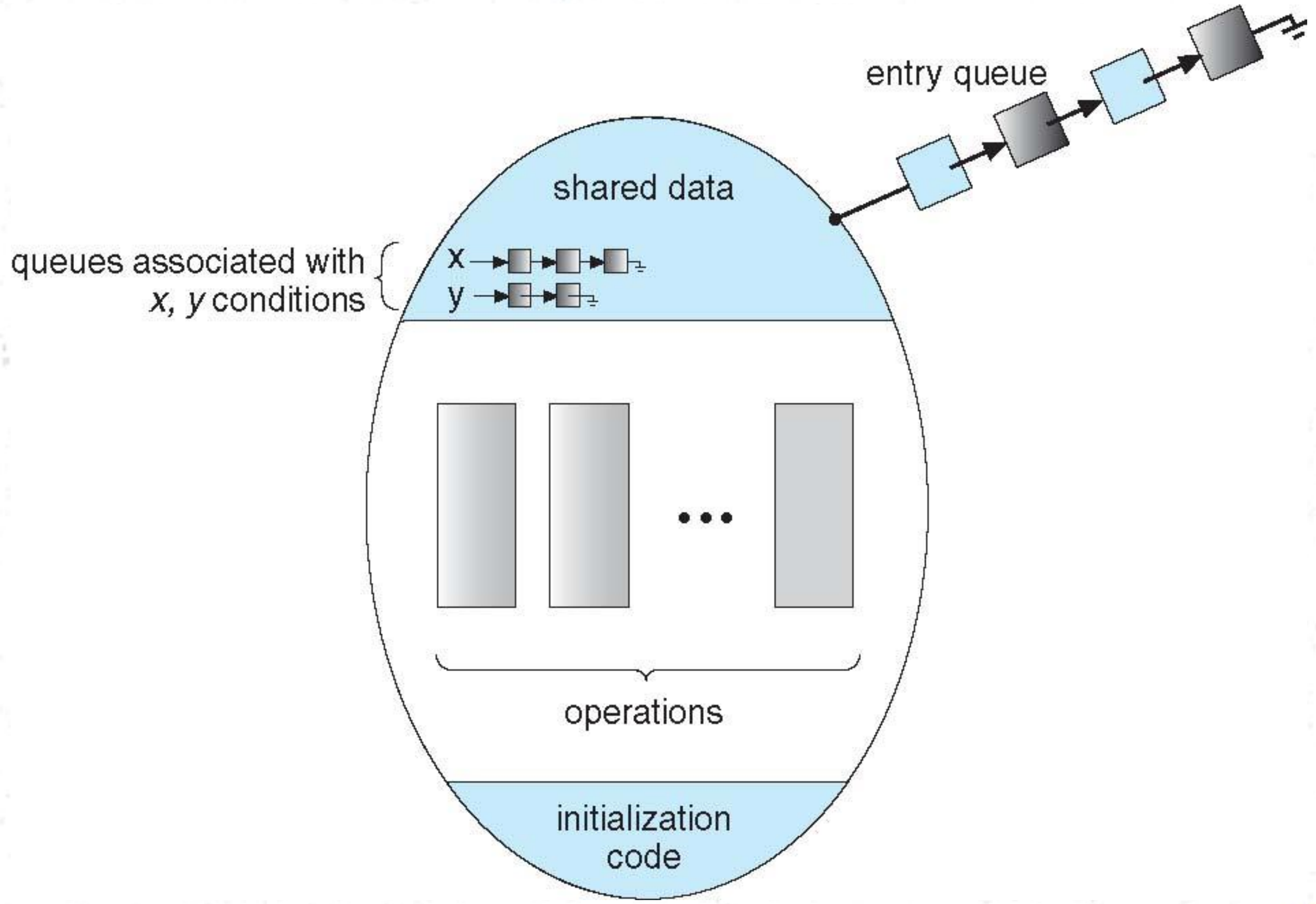
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- But not powerful enough to model some synchronization schemes

```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…) { …. }

  procedure Pn (…) {……}

     Initialization code (…) { … }
  }
}
```

Schematic view of a Monitor



Monitor with Condition Variables



End of Lecture 6